CSE 341 AB: Section 8

Josh Pollock

Office Hours: Tuesdays 3:00pm - 4:00pm

Questions?

Agenda

Array: Ruby’s Sequence Workhorse

Hashes: Dynamic Records

Ranges: The Power of Enumerators

Ruby Closures
- Blocks, Procs, and Lambdas

Ruby Under a Magnifying Glass
- Objects
- Classes

Array: Ruby’s Sequence Workhorse

The Many (Inter)Faces of the Ruby Array

Ruby uses dynamically sized arrays like Java's ArrayLists.

These are nice middle ground between linked lists and statically sized arrays.

Allow fast random access and asymptotically fast insertion and deletion.

Ruby array entries don't need to have the same type
("natural" in dynamically typed languages)

Ruby arrays are super *flexible*.

Ruby uses arrays for lists, sets, stacks, and queues!

(Ordered) Sets

We can model sets as arrays without duplicate entries.

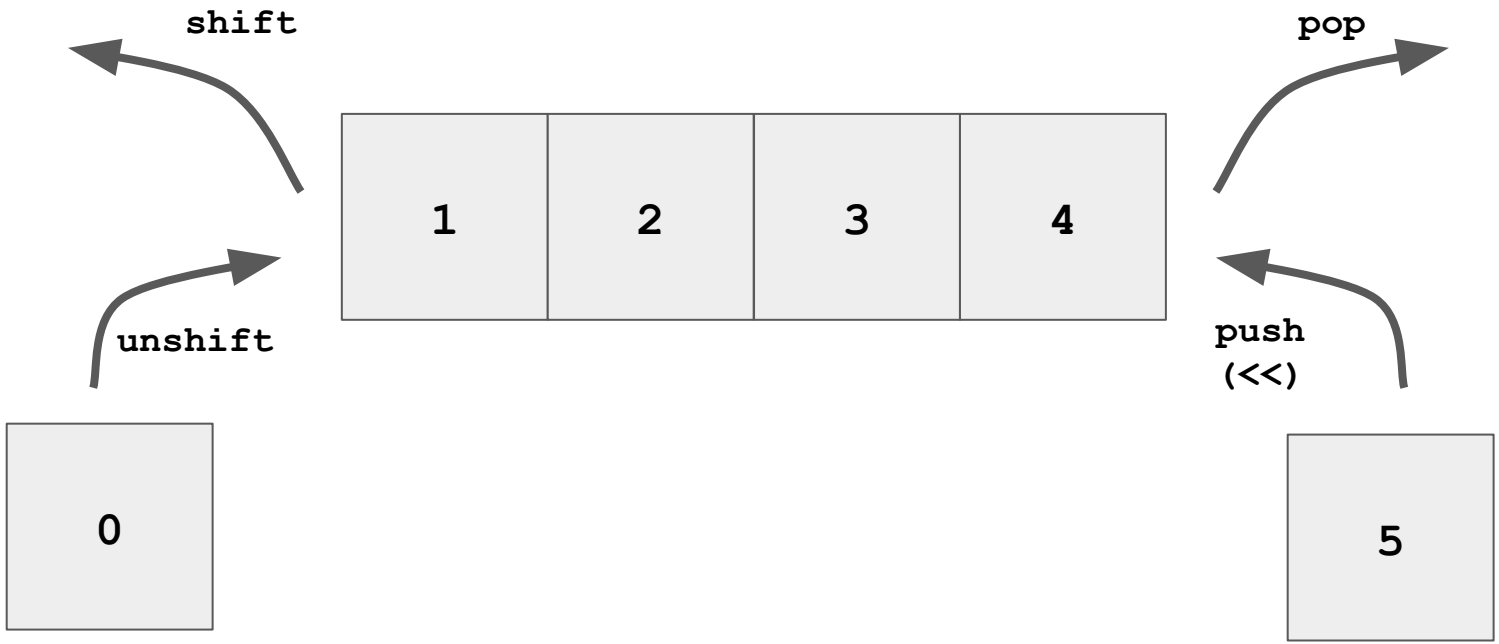
Remove duplicates with `.uniq`

Union: `|`.

Intersection: `&`.

Difference: `-`.

Stacks and Queues



To the Code!

(examples and more useful methods)

Hashes: Dynamic Records

A map from keys to values.

Keys don’t have to have the same type!

Keys and entries are mutable. They can be updated dynamically.

See code for examples.

Ranges: The Power of Enumerators

Ranges are *enumerators*, not lists.

Somewhat like the streams we saw in Racket, they are lazy.

The only do computation when necessary.

Syntax:

`i..j` [i, j] -- includes j

`i...j` [i, j) -- excludes j

For step size, use `.step`.

The Takeaway

Ruby has several flexible ways of constructing complex data.

This flexibility is characteristic of dynamically typed languages (cf. Python).

Consult the Ruby documentation. It’s really good.

Ruby Closures

Let’s Take a Closer Look at Ruby Closures

Ruby gives us 3 ways to define a closure:

- Block
- Proc
- Lambda

Lexical scope, but variables are stored as references to objects.
E.g. Modifying an array referenced by a closure may change its behavior.

Use `.call` to call them.

Block Cheat Sheet

The most common type of closure in Ruby.

All methods take a block argument, it may not be used.

Call a block with **`yield`**.

Use **`return`** to return from an enclosing method.

Give a block an explicit name with **`&block_name`**.

Proc

Procs are essentially blocks as objects.

Initialize like any other object.

Issues with Blocks and Procs

`return` jumps out of the method where the block was called.

They don’t check they’re passed the right number of arguments.

Lambda

Lambda is a special kind of Proc with special behavior.

Create with **`lambda`** or **`->`**.

Work like “normal” closures.

`return` returns from the lambda.

Lambda checks it gets the right number of arguments.

Diving In

If I were to write an explanation of Ruby closures, I would mostly just write this:

https://www.rubyguides.com/2016/02/ruby-procs-and-lambdas/

So let’s just use it!

Practice Using Blocks

Let's write **`Array#map`**.

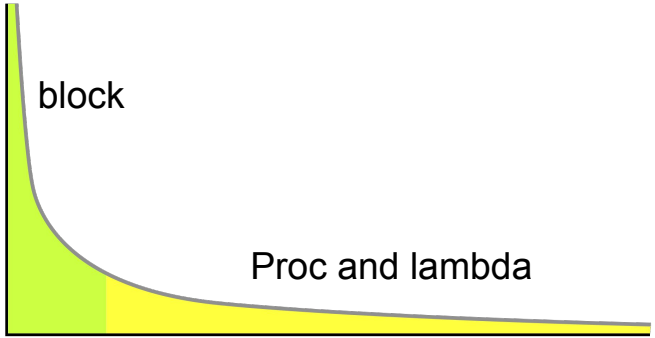
The Takeaway

Ruby takes a pragmatic, OO approach to first-class functions.

The typical case is supported by **blocks**. You should use them most often.

Ruby is a real-word language so it supports the long-tail of use cases with Proc and lambda.

This makes the language more complex, especially b/c Proc and lambda extend the language *implementation*.



https://en.wikipedia.org/wiki/Long_tail#/media/File:Long_tail.svg

Ruby Under a Magnifying Glass*

*An oversimplified adaptation of
Chapter 5 from *Ruby Under a Microscope* by Pat Shaughnessy

Ruby’s Data Representation

Everything really is an object!

But there is a special type of object: a class definition.

“Primitives” like integers, strings, and arrays also have their own special representations, but we can ignore these subtleties.

What Is a Ruby Object (Roughly)?

Just a pair of a pointer to a class and a map from instance variable names to values.

```
class Point
  attr_accessor :x, :y
  def initialize(x, y)
    @x = x
    @y = y
  end
end

class ColorPoint < Point
  attr_accessor :color
  def initialize(x, y, color)
    super(x, y)
    @color = color
  end
end
```

```
cp = ColorPoint.new(0, 0, "red")
```

cp

class **ColorPoint** (really points to a class object)	instance variables (@)	
	x	**0**
	y	**0**
	color	**"red"**

What Is a Ruby Class (Roughly)?

It must be an object so it has to have a class pointer and instance variables.

```
class Point
  attr_accessor :x, :y
  def initialize(x, y)
    @x = x
    @y = y
  end
end

class ColorPoint < Point
  attr_accessor :color
  def initialize(x, y, color)
    super(x, y)
    @color = color
  end
end
```

ColorPoint

class **Class**	class instance variables (`@`)

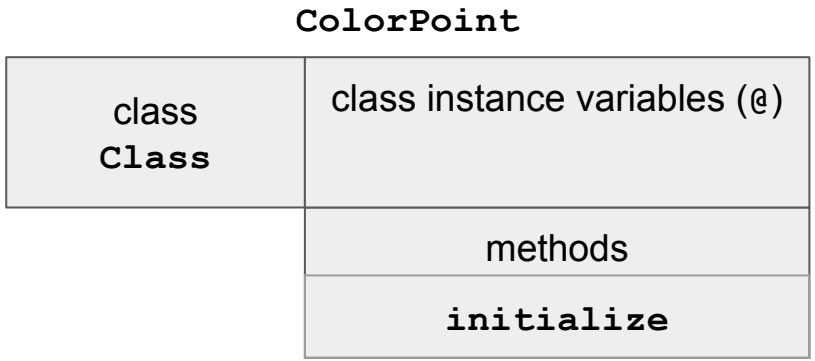
A class instance variable is created by assigning to an instance variable in the class's top-level scope.

What Is a Ruby Class (Roughly)?

But it also needs a list of methods.

```
class Point
  attr_accessor :x, :y
  def initialize(x, y)
    @x = x
    @y = y
  end
end

class ColorPoint < Point
  attr_accessor :color
  def initialize(x, y, color)
    super(x, y)
    @color = color
  end
end
```

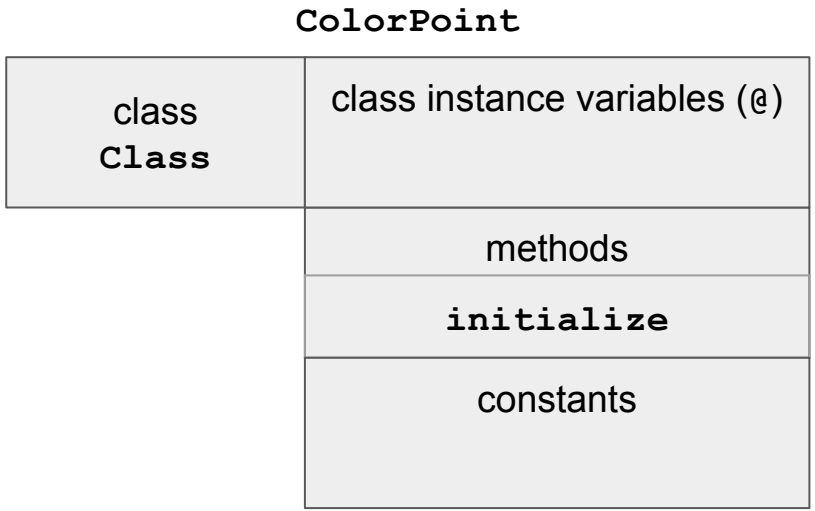


What Is a Ruby Class (Roughly)?

And constants...

```
class Point
  attr_accessor :x, :y
  def initialize(x, y)
    @x = x
    @y = y
  end
end

class ColorPoint < Point
  attr_accessor :color
  def initialize(x, y, color)
    super(x, y)
    @color = color
  end
end
```

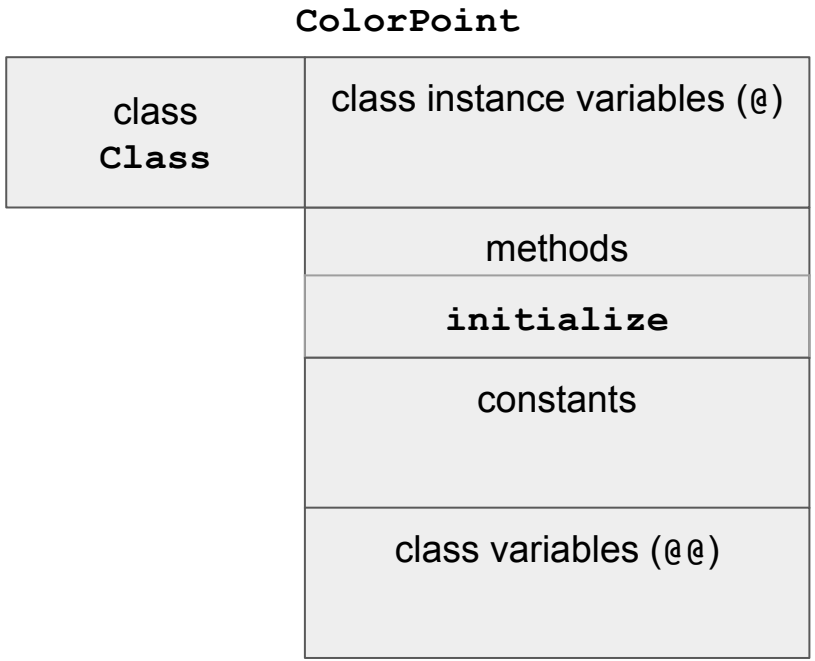


What Is a Ruby Class (Roughly)?

And class variables!

```
class Point
  attr_accessor :x, :y
  def initialize(x, y)
    @x = x
    @y = y
  end
end

class ColorPoint < Point
  attr_accessor :color
  def initialize(x, y, color)
    super(x, y)
    @color = color
  end
end
```

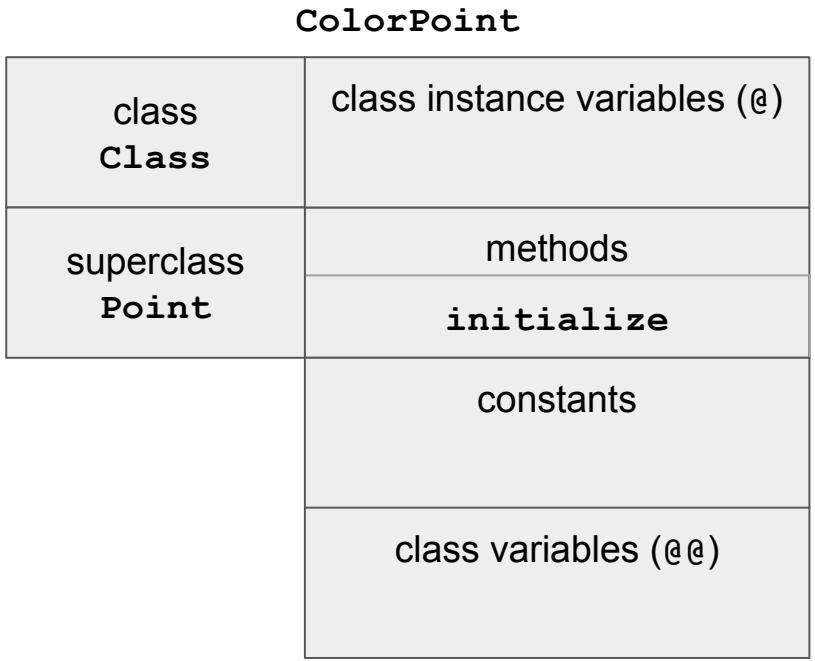


What Is a Ruby Class (Roughly)?

Oh and a superclass.

```
class Point
  attr_accessor :x, :y
  def initialize(x, y)
    @x = x
    @y = y
  end
end

class ColorPoint < Point
  attr_accessor :color
  def initialize(x, y, color)
    super(x, y)
    @color = color
  end
end
```



What Is a Ruby Class (Roughly)?

We haven’t even touched on class methods! There are still a bunch of lies here.

```
class Point
  attr_accessor :x, :y
  def initialize(x, y)
    @x = x
    @y = y
  end
end

class ColorPoint < Point
  attr_accessor :color
  def initialize(x, y, color)
    super(x, y)
    @color = color
  end
end
```

ColorPoint

class **Class**	class instance variables (@)
superclass **Point**	methods
	initialize
	constants
	class variables (@@)

The Takeaway

Ruby has a more complicated runtime model than other languages we've seen.

This model is *leaky*! We need to understand some aspects of the runtime.

Classes "own" methods and constants.

A class has a class pointer since it's an object.

A class has a superclass pointer since it's a class.

Object

class pointer	instance variables (`@`)

Class

class pointer	class instance variables (`@`)
superclass pointer	methods
	constants
	class variables (`@@`)